Programmeerimise algkursus

I200

Loengu sisu

- Meeldetuletus: muutujad, tüübid, if ja while
- If: lisavõimalused. Case.
- While: lisavõimalused. For, do. Break, continue.
- Massiivid.

Programmi kirjutamise etapid:

- **getting the program text into the computer,**
- **compiling the program, and**
- **running the compiled program.**

Final step - running the program - either as

- **Application** - program running without a www browser

- Applet- program running in a www browser
- Servlet- program running in a (web) server

Java: APPLICATION

```
public class HelloWorld {

   // A program to display the message
   // "Hello World!" on standard output

   public static void main(String[] args) {
        System.out.println("Hello World!");
   }

}    // end of class HelloWorld
```

- kompileerime: **javac HelloWorld.java =>** HelloWorld.class
- paneme käsurealt käima: **java HelloWorld**

function called **main**, with a definition that takes the form:

```
   public static void main(String[] args) {
      statements
   }
```

Variables & primitive names

- `N n rate x15 a_long_name time_is_$ HelloWorld`

- `HelloWorld, helloworld, HELLOWORLD, and hElloWorLD are all distinct names`

- reserved words include: `class, public, static, if, else, while, and several dozen other words.`

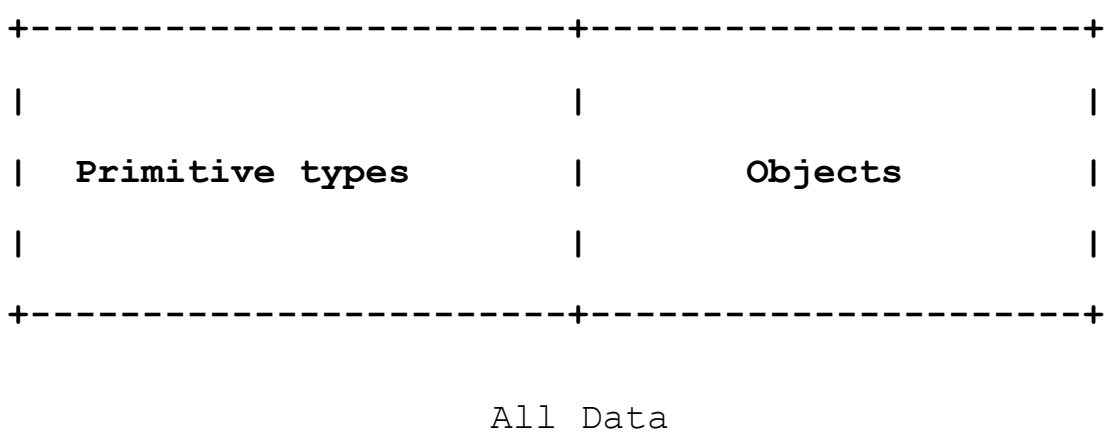
- An assignment statement takes the form:

  variable = expression;

```
rate = 0.07;
interest = rate * principal;
```

Data

- Java has *very many* data types built into it, and you (as a programmer) can create as many more as you want.
- However, other than the primitive data types, *all the other data in a Java program will be represented as an object*. So there is a fundamental split in the data a Java program deals with:



Primitive TYPES

- The primitive types are named:

```
byte, short, int, long,
float, double,
char,
boolean
```

- **short** corresponds to two bytes (16 bits). Variables of type short have values in the range -32768 to 32767.
- **int** corresponds to four bytes (32 bits). Variables of type int have values in the range -2147483648 to 2147483647.
- **long** corresponds to eight bytes (64 bits). Variables of type long have values in the range -92233720368547758O8

```
boolean result: rate > 0.05
```

String: "I said, \"Are you listening!\"\n"

Variable assignment

type-name variable-name;

or

type-name variable-name = expression;

```
int N;
double x;
double rate = 0.07;
char space = ' ';
```

You can create several variables in the same declaration, if you separate them by commas. For example:

```
double x,y;
char first = 'D',
     middle = 'J',
     last = 'E';
int i, j = 17;
```

Interest calculation

```
public class Interest {
 public static void main(String[] args) {
     // the value of the investment
     double principal = 17000;
     // the annual interest rate
     double rate = 0.07;
     // interest earned in one year
     double interest;

    interest = principal * rate;
    principal = principal + interest;

    System.out.print
     ("The interest earned is $");
    System.out.println(interest);
    System.out.print
      ("The value of the investment after one year is $");
    System.out.println(principal);

  } // end of main()
} // end of class Interest
```

Block

- The block is the simplest type of statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is:

```
{
    statements
}
```

- Here are two examples of blocks:

```
{
  System.out.print("The answer is ");
  System.out.println(ans);
}
```

```
{  // This block exchanges the values of x and y
   int temp = x;  // declare temp and store x in it
   x = y;         // copy value of y into x
   y = temp;      // copy value of temp into y
}
```

IF statement

- An if statement has one of the forms:

```
if ( boolean-expression )
    statement
else
    statement
```

```
if ( boolean-expression )
    statement
```

- As usual, each of the **statement**'s in an if statement can be a block, so that an if statement often looks like:

```
if ( boolean-expression ) {
    statements
}
else {
    statements
}
if ( boolean-expression ) {
    statements

}
```

IF statement

```
if ( x > y ) {
  int temp = x;   // declare temp and store x
  x = y;          // copy value of y into x
  y = temp;       // copy value of temp into y
}
```

Finally, here is an example of an if statement that includes an `else part`.

```
if ( years > 1 ) {

   System.out.print

   ("The value of the investment after ");

   System.out.print(years);
   System.out.print(" years is $");
}
else {  // handle case for 1 year
   System.out.print
    ("The value of the investment after 1 year is $");
}  // end of if statement
System.out.println(principal);
```

While LOOP

- A while loop has the form:

```
while (boolean-expression)
     statement
```

- Since the statement can be, and usually is, a block, many while loops have the form:

```
while (boolean-expression) {
     statements
}
```

While LOOP

- Here is an example of a while loop that simply prints out the numbers 1, 2, 3, 4, 5:

```
int number = 1;
while ( number < 6 ) {
      System.out.println(number);
      number = number + 1;
}
System.out.println("Done!");
```

if-else ladder

- Example: not two, but three cases to check:

```
if (boolean-expression-1)
     statement-1
 else
     if (boolean-expression-2)
        statement-2
     else
        statement-3
```

- Normally written as:

```
if (boolean-expression-1)
   statement-1
else if (boolean-expression-2)
   statement-2
else
   statement-3
```

if-else ladder

- Here is an example that will print out one of three different messages, depending on the value of a variable named temperature:

```
if (temperature < 50)
    System.out.println("It's cold.");
 else if (temperature < 80)
    System.out.println("It's nice.");
 else
    System.out.println("It's hot.");
```

if-else ladder

- You can go on stringing together "**else-if's**" to make multi-way branches with any number of cases:

```
if (boolean-expression-1)
     statement-1
 else if (boolean-expression-2)
     statement-2
 else if (boolean-expression-3)
     statement-3
        .
        . // (more cases)
        .
 else if (boolean-expression-N)
     statement-N
 else
     statement-(N+1)
```

if-else in first branch danger!!

- Check that:

```
if ( x > 0 )
   if (y > 0)
       System.out.println("First case");
else
   System.out.println("Second case");
```

- . . . is probably misunderstood

a switch as a special form of if-else ladder

- A **switch** statement has the form:

```
switch (expression) {
    case constant-1:
        statements-1
        break;
    case constant-2:
        statements-2
        break;
      .
      .   // (more cases)
      .
    case constant-N:
        statements-N
        break;
    default:  // optional default case
        statements-(N+1)
    } // end of switch statement
```

switch example

```
switch (N) {   // assume N is an integer variable
    case 1:
       System.out.println("The number is 1.");
       break;
     case 2:
     case 4:
     case 8:
       System.out.println("The number is 2, 4, or 8.");
       System.out.println("(That's a power of 2!)");
       break;
     case 3:
     case 6:
     case 9:
       System.out.println("The number is 3, 6, or 9.");
       System.out.println("(That's a multiple of 3!)");
       break;
     case 5:
       System.out.println("The number is 5.");
       break;
     default:
       System.out.println("The number is 7,");
       System.out.println(" or outside the range 1 to 9");
  }
```

While LOOP

- A while loop has the form:

```
while (boolean-expression)
     statement
```

- Since the statement can be, and usually is, a block, many while loops have the form:

```
while (boolean-expression) {
     statements
}
```

While LOOP

- Here is an example of a while loop that simply prints out the numbers 1, 2, 3, 4, 5:

```
int number = 1;
while ( number < 6 ) {
      System.out.println(number);
      number = number + 1;
}
System.out.println("Done!");
```

Do…while loop

- A do…while loop has the form:

```
do
    statement
while (boolean-expression);
```

- Since the statement can be, and usually is, a block, many do…while loops have the form:

```
do  {
      statements
} while (boolean-expression);
```

Do…while loop example

- Pseudocode:

```
do {
     Play a Game
     Ask user if he wants to play another game
     Read the user's response
} while ( the user's response is yes );
```

- Real code:

```
boolean wantsToContinue;  // True if user wants to play
                          // again.
do {
    Checkers.playGame();
    TextIO.put("Do you want to play again? ");
    wantsToContinue = TextIO.getlnBoolean();
 } while (wantsToContinue == true);
```

Break and continue

- The syntax of the while and do..while loops allows you to test the continuation condition at either the beginning of a loop or at the end.
- Sometimes, it is more natural to have the test in the middle of the loop, or to have several tests at different places in the same loop.
- Java provides a general method for breaking out of the middle of any loop. It's called **the break statement**, which takes the form

```
break;
```

- When the computer executes a break statement in a loop, it will immediately jump out of the loop. It then continues on to whatever follows the loop in the program.

Example: break

```
while (true) {  // looks like it will run forever!
    TextIO.put("Enter a positive number: ");
    N = TextIO.getlnlnt();
    if (N > 0)   // input is OK; jump out of loop
       break;
    TextIO.putln("Your answer must be > 0.");
}
// continue here after break
```

- A break statement terminates the loop that immediately encloses the break statement. It is possible to have nested loops, where one loop statement is contained inside another. If you use a break statement inside a nested loop, it will only break out of that loop, not out of the loop that contains the nested loop.

- There is a "labeled break" statement that allows you to specify which loop you want to break.

Continue

- The continue statement is related to break, but less commonly used. A continue statement tells the computer to skip the rest of the current iteration of the loop.

- However, instead of jumping out of the loop altogether, it jumps back to the beginning of the loop and continues with the next iteration (after evaluating the loop's continuation condition to see whether any further iterations are required).

For LOOP

- The for statement makes a common type of while loop easier to write.

- Many while loops have the general form:

```
initialization
 while ( continuation-condition ) {
        statements
        update
 }
```

For LOOP

- Example with while:

```
years = 0;  // initialize the variable years
 while ( years < 5 ) {   // condition for continuing loop
    interest = principal * rate;
    principal += interest;
    System.out.println(principal);
    years++;   // update the value of the variable, years
}
```

- This loop can be written as the following equivalent for statement:

```
for ( years = 0;  years < 5;  years++ ) {
    interest = principal * rate;
    principal += interest;
    System.out.println(principal);
 }
```

For LOOP

- The formal syntax of the for statement is as follows:

```
for ( initialization; continuation-condition; update )
    statement
```

- or, using a block statement:

```
for ( initialization; continuation-condition; update ) {
    statements
}
```

- The continuation-condition must be a boolean-valued expression. The initialization can be any expression, as can the update.
- Any of the three can be empty. If the continuation condition is empty, it is treated as if it were "true," so the loop will be repeated forever or until it ends for some other reason, such as a break statement.
- Some people like to begin an infinite loop with "for (;;)“ instead of "while (true)".

Nested loops

- How to print a multiplication table like follows?

```
 1   2   3   4   5   6   7   8   9  10  11  12
 2   4   6   8  10  12  14  16  18  20  22  24
 3   6   9  12  15  18  21  24  27  30  33  36
 4   8  12  16  20  24  28  32  36  40  44  48
 5  10  15  20  25  30  35  40  45  50  55  60
 6  12  18  24  30  36  42  48  54  60  66  72
 7  14  21  28  35  42  49  56  63  70  77  84
 8  16  24  32  40  48  56  64  72  80  88  96
 9  18  27  36  45  54  63  72  81  90  99 108
10  20  30  40  50  60  70  80  90 100 110 120
11  22  33  44  55  66  77  88  99 110 121 132
12  24  36  48  60  72  84  96 108 120 132 144
```

Nested loops

- Example code with one for loop inside another

```
for ( rowNumber = 1;  rowNumber <= 12;  rowNumber++ ) {
   for ( N = 1;  N <= 12;  N++ ) {
      // print in 4-character columns
      TextIO.put( N * rowNumber,  4 );
   }
   TextIO.putln();
}
```

Built-in arithmetic and logic statements

- Standard arithmetics:

```
x = x + 2;
x = x * 2;
x = x / 2;    // gives integer for int args
x = x % 2;    // gives remainder
```

- Shorthand versions:

```
x++; // adds 1: means x = x + 1; results with
  // the old value (before adding)
++x; // like x++, but results with the new value
x--; // analogous: x = x - 1;
--x; // analogous: x = x - 1;
x +=x; x *= 2; x /= 2; x %=2;    // shorthand
    //versions for x = x + 2; etc
```

More built-in statements

- logic:

```
x = y && z; // x becomes truth value (y and z)
  // eg: ((1<10) && (2<3)) gives true
x = y || z; // x becomes truth value (y or z)
  // eg: ((100<10) || (2<3)) gives true
!x; // gives negation (true->false, false->true
  // eg:  !(1<2) is true
```

Empty statement

- You can write extra semicolons if you like: these are treated as empty statements.

```
if (x < 5)
  System.out.println("Hello"); ; ;
; x = x + 4 ; ;
```

is perfectly OK code

- Beware!

```
for (int i = 0; i < 10; i++);
    System.out.println("Hello");
```

does NOT print “Hello” 10 times!!

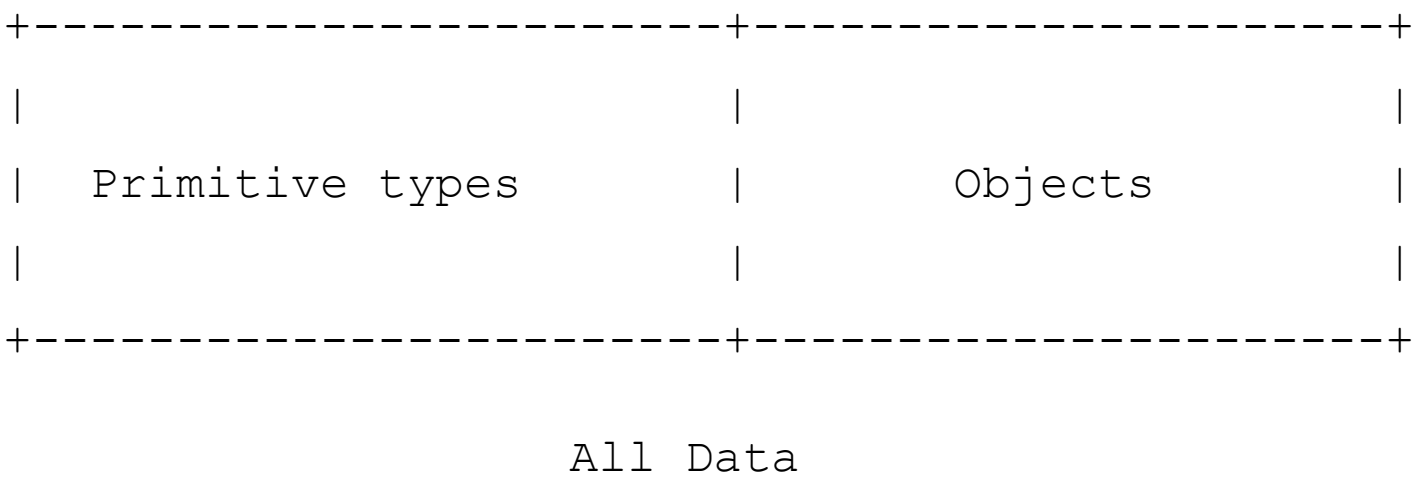
Statement types

- **declaration statement (for declaring variables)**
- **assignment statement**
- **subroutine call statement (including input/output routines)**
- other expression statement (such as "x++;")
- empty statement
- **block statement**
- **while statement**
- do..while statement
- **if statement**
- **for statement**
- switch statement
- break statement (found in loops and switch statements only)
- continue statement (found in loops only)
- **return statement (found in subroutine definitions only)**
- try..catch statement
- throw statement
- synchronized statement

Data

- Java has *very many* data types built into it, and you (as a programmer) can create as many more as you want.

- However, other than the primitive data types, *all the other data in a Java program will be represented as an object*. So there is a fundamental split in the data a Java program deals with:



Primitive TYPES

- The primitive types are named:

```
byte, short, int, long,
float, double,
char,
boolean
```

- short corresponds to two bytes (16 bits). Variables of type short have values in the range -32768 to 32767.
- int corresponds to four bytes (32 bits). Variables of type int have values in the range -2147483648 to 2147483647.
- long corresponds to eight bytes (64 bits). Variables of type long have values in the range

  -9223372036854775808 to 9223372036854775807.

- `boolean is result of: rate > 0.05`

What is a complex type?

- Basically, a complex type is a place to store several simple types, one after another. You can then use this “data object” as one whole.

- Examples:

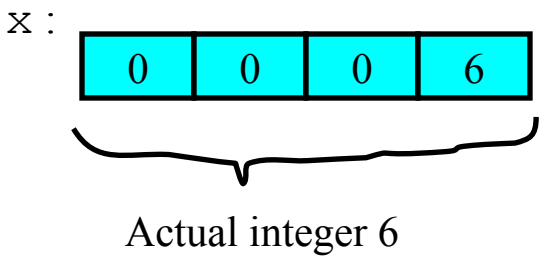
  - String: several simple characters one after another: “asasad adsas”
  - Array: like a string, but may contain integers, floats or any other type of data (also complex types)!
  - A structure representing a person: contains, for example
    - A string holding the name: “Jaan Mets”
    - An integer holding age: 21
    - A float holding length: 1.83

Where are complex objects in memory?

- If you use a simple type, like integer:

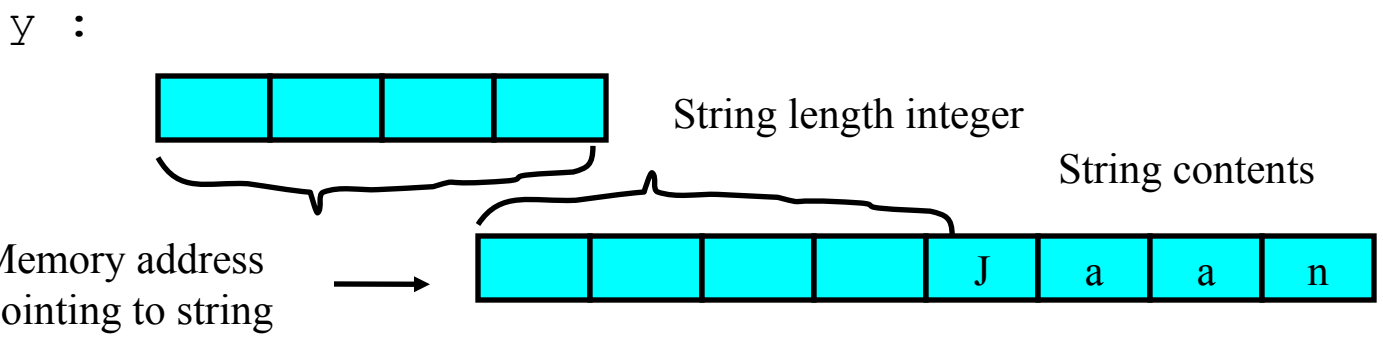
```
int x = 6;
```

then this "x" represents 4 bytes in memory holding value 6.



- If you want to create a string "Jaan Mets", it will also use up some memory and it will be placed at a some address (for example, 23423432 in memory)

```
String y = "Jaan";
```



Variables and objects in memory

- To summarise:
  - Variables of simple type hold the value directly
  - Variables of complex type hold a pointer to the value
  - The complex object is placed by the system somewhere in memory

- Therefore:
  - Declaring a simple type variable (int x) automatically creates place of value in memory
  - Declaring a complex type variable (String y) only creates a pointer
  - To actually create an object itself in memory one must say `new`

```
String y;
y = new String("Jaan Mets")
```

Important notes

- The arguments to `new` are different for different types
- You can always give a special value `null` to variable of a complex type
- While `new` is similar to `malloc` in C, you never need to say "`free`":
  - Useless objects in memory are thrown away automatically by Java
  - Object is useless, if no variables point to it

Example:

```
String y = new String("Jaan Mets");
y = null;
```

  - Automatic throwing away of objects in memory is called "Garbage collection" and is a part of most modern languages

Strings and arrays are somewhat special

- Strings and arrays are both just objects.
- However:
  - Since they are very common, there is a simplified way to create them.
  - There are many “built-in” methods and functions for both.
  - Array elements are accessed by a standard array syntax, like in C.
- NB! Differently from C:
  - A string in Java is not just a an array of characters.
  - The length of arrays and string is always kept along with contents

Array

- An array is just a sequence of items.
- The items in an array are numbered, and individual items are referred to by their position number.
- All the items in an array must be of the same type. So, the definition of an array is a numbered sequence of items, which are all of the same type.
- The number of items in an array is called the length of the array. The position number of an item in an array is called the index of that item. The type of the individual elements in an array is called the base type of the array.

- Java arrays are objects.
  - Arrays are created using the `new` operator.
  - No variable can ever hold an array; a variable can only refer to an array.
  - Any variable that can refer to an array can also hold the value `null`, meaning that it doesn't at the moment refer to anything.
  - An array belongs to a class, which like all classes is a subclass of the class `Object`.

Array

- Suppose that `A` is a variable that refers to an array.
  - The first item is `A[0]`,
  - the second is `A[1]`,
  - … and so forth.

- `A[k]` can be used just like a variable.

  You can assign values to it, you can use it in expressions, and you can pass it as a parameter to subroutines.

- Syntax:

  array-variable [integer-expression]

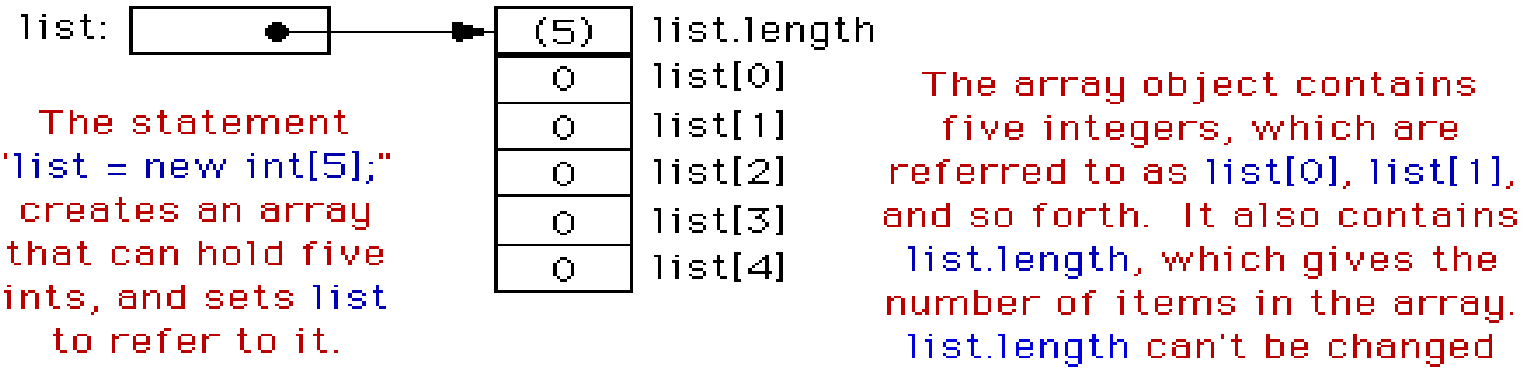
  for referring to an item in an array.

Array creation

- `int[] list;`

  creates a variable named `list` of type `int[]`.
- This variable is capable of referring to an array of `int`s, but initially its value is `null`
- The `new` operator is used to create a new array object, which can then be assigned to `list`.
- An example:

```
list = new int[5];
```



Array creation, elements

- Newly created array of integers is automatically filled with zeros.
- New array is always filled with a known, default value:
  - zero for numbers,
  - `false` for boolean,
  - the character with Unicode number zero for `char`,
  - `null` for objects.

- And the following loop would print all the integers in the array:

```
for (int i = 0; i < list.length; i++) {
     System.out.println( list[i] );
}
```